Laboratorium Informatyki II
Ćwiczenie 4

1 Przeciążanie funkcji i operatorów

W języku C++ możliwe jest zdefiniowanie wielu funkcji o tej samej nazwie oraz nadanie dostępnym operatorom nowych właściwości. Procedura ta określana jest jako **przeciążenie**. Procedura ta jest dostępna zarówno przy tworzeniu nowych elementów programu jak i klasy. Poszczególne implementacje funkcji lub metody mogą wykonywać analogiczny algorytm ale dedykowanych dla rożnych typów danych lub realizować całkiem odmienne zadania. Identyfikacja odbywa się na podstawie liczby i typów podawanych atrybutów. Przeciążenie funkcji lub metody nie może zostać zrealizowane gdy różnią się tylko typem zwracanej wartości. Poniżej pokazano kod z implementacja przeciążenia funkcji.

```
#include <iostream>
using namespace std;
int suma(int,int);
double suma(double, double);
string suma(string, string);
int main()
{
    int A = suma(4,3);
    double B = suma(4.0,3.0);
    string C = suma("Pierwszy","Drugi");
    string D = suma("Trzeci","Czwarty");
    cout << "A=" << A << " --> B=" << B << endl;
    cout << "C=" << C << " --> D=" << D << endl;
}
int suma(int x, int y){
    return x+y;
}
double suma(double x, double y){
    return x+y;
}
string suma(string x, string y){
    string s = x + "<->" + y;
    return s;
}
```

Przy projektowaniu klasy w języku C++ można zaprogramować przeciążenie operatora poprzez nadanie mu nowej funkcjonalności realizowanej na obiektach danej klasy. procedura jest analogiczna do przeciążenia funkcji i metod. Przeciążenie operatora można zrealizować także poza kodem klasy.

Przeciążenie operatora realizowane jest za pomocą instrukcji `operator`. Typem zwracanym jest zazwyczaj obiekt klasy, podobnie jak atrybutu na jakich operator działa. Możliwe jest jednak zaprogramowanie działania operatora tak aby zwracał i przyjmował wartości innych typów. Poniżej pokazano projekt klasy z zaprogramowanymi przeciążeniami operatorów.

```
#include <iostream>
using namespace std;
class dane{
public:
    dane(){
        x=0;y=0;
    }
    dane(int x, int y){
        this->x=x; this->y=y;
    }
    void drukuj(){
        cout << "(" << x <<"," << y << ")" << endl;
    }
    bool operator == (const dane &D){
        if(this->x==D.x && this->y==D.y)
            return true;
        else return false;
    }
    dane operator *(const int &D){
        dane temp(0,0);
        temp.x=this->x*D;
        temp.y=this->y*D;
        return temp;
    }
    dane operator +(const dane &D){
        dane temp(0,0);
        temp.x=this->x+D.x;
        temp.y=this->y+D.y;
        return temp;
    }
private:
    int x,y;
};
int main()
{
    dane A(1,2), B(4,5), C;
    C.drukuj(); A.drukuj(); B.drukuj();
    C = A + B; C.drukuj();
    dane D(5,7);
    if(D==C)
        cout << "Ten sam punkt" << endl;
    else
        cout << "Inne punkty" << endl;
    D = D * 4; D.drukuj();
}
```

W przypadku operatora * należy zwrócić uwagę że nie zapewnia on przemienności działania. Konieczne by było zaprogramowanie przeciążenia operatora mnożenia poza kodem klasy, jak pokazano poniżej.

```
dane operator *(const int &A, const dane &B){
    dane temp(0,0);
        temp.x=A*B.x;
        temp.y=A*B.y;
        return temp;
}
```

Taka implementacja wymaga aby cechy klasy `x` i `y` były `public:` lub dodanie do klasy odpowiednich metod odczytania i ustawienia cech prywatnych. Alternatywą może być nadanie przyjaźni pomiędzy klasą a funkcją przeciążającą operator `*`.

2 Przyjaźń

Przyjaźń jest sposobem na nadanie dostępu do elementów prywatnych klasy funkcjom lub innym klasom. Przyjaźń należy zdefiniować w klasie która chce udostępnić swoje zasoby elementom zewnętrznym. Przyjaźń nadaje się poprzedzając deklarację funkcji lub klasy słowem kluczowym `friend`, jak pokazano poniżej:

```
friend class nazwa;
friend void funkcja(...);
friend int operator +(...);
```

W praktyce sprowadza się to do dodania w kodzie klasy w obszarze publicznym deklaracji klasy lub funkcji poprzedzonej słowem `friend`. Klasa może być zaprzyjaźniona z wieloma funkcjami i klasami, podobnie funkcja może być zaprzyjaźniona z wieloma klasami.

3 Dziedziczenie

Dziedziczenie jest procedurą rozszerzenia funkcjonalności klasy o cechy i metody innej klasy. Metody i cechy klasy "rodzica" stają się metodami i cechami klasy "potomka". Dziedziczenie zapewnia wygodny mechanizm ponownego wykorzystania kodu.

Nie wszystkie elementy klasy "rodzica" są dziedziczone, nie podlegają temu konstruktory i destruktor oraz operator przypisania. Stopnień dziedziczenia jest określany analogicznie jak dostępność do elementów klasy i określa zakres elementów jakie są dziedziczone. Klasa może dziedziczyć tylko elementy z poziomu publicznego (`public:`) i chronionego (`protected:`). Poziom dziedziczenia określa na jakiej zasadzie dostępne będą dziedziczone elementy w kodzie programy przy odwołaniu do obiektu.

```
#include <iostream>
using namespace std;
class dane{
    public:
        int x;
        dane(){
            x=1;y=2;z=3;
        }
        void drukuj(){
```

```
            cout << "(" << x << ","<< y << "," << z << ")" << endl;
        }
    private:
        int y;
    protected:
        int z;
};
class nazwa: public dane{
    public:
        nazwa(){
            name = "test";
            x = 10;
            //y = 20; cecha niedostępna
            z = 30;
        }
        void drukuj(){
            cout << name << "(" << x << "," << z << ")" << endl;
            cout << name; dane::drukuj();
        }
    private:
        string name;
};
int main()
{
    nazwa obiekt;
    obiekt.drukuj();
    cout << "x=" << obiekt.x << endl;
}
```

4 Polimorfizm

Polimorfizm inaczej określany jako wielopostaciowość, jest rozumiany możliwość przyjmowania przez funkcje i metody rożnych form. Rozróżnia się polimorfizm statyczny i dynamiczny. Pod pojęciem polimorfizmu statycznego rozumie się przeciążanie funkcji, metod i operatorów.

Polimorfizm dynamiczny jest zagadnieniem z związanym z stosowanym w dziedziczeniu wyabstrahowaniem metody od klasy. Powiązane z tym są klasy abstrakcyjne i metody wirtualne. W standardowym dziedziczeniu metody z klasy podstawowej są przesłaniane przez metody z klasy dziedziczącej. W polimorfizmie zachodzi sytuacja odwrotna. Poprzez wirtualna metodę z kasy podstawowej realizowane jest odwołanie do metod z klas dziedziczących. Realizowane jest to przez specjalną klasę bazową zwaną abstrakcyjną. Cechuje się ona tym że jest pusta, nie realizuje żadnego kodu, nie tworzy się na jej podstawie obiektu, a specjalny wskaźnik zwany polimorficznym. Poprzez ten wskaźnik odwołuje się do metod wirtualnych. Przykład implementacji polimorfizmu dynamicznego pokazano na kodzie pokazanym poniżej:

```
plik main.cpp:
#include <iostream>
#include "klasy.hpp"
using namespace std;
```

```
int main()
{
    figura *kw = new kwadrat(5);
    figura *kol = new kolo(2.5);
    figura *tr = new trojkat(5,5);
    figura *ka[3] = {kw,kol,tr};
    for(int i=0;i<3;i++){
        cout << "Figura-->"; ka[i]->name();
        cout << "\tPole: " << ka[i]->pole() << endl;
        cout << "\tObwód: " << ka[i]->obwod() << endl;
    }
}
```

---

plik klasa.hpp:

```
#ifndef KLASY_HPP
#define KLASY_HPP
#include <iostream>
using namespace std;
class figura
{
public:
    virtual float pole();
    virtual float obwod();
    virtual void name();
};
class kolo : public figura
{
private:
    float r;
    string nazwa;
public:
    kolo(float);
    virtual float pole();
    virtual float obwod();
    virtual void name();
};
class kwadrat : public figura
{
private:
    float a;
    string nazwa;
public:
    kwadrat(float);
    virtual float pole();
    virtual float obwod();
    virtual void name();
};
class trojkat : public figura{
private:
    float a,h;
```

```
    string nazwa;
public:
    trojkat(float,float);
    virtual void name();
    virtual float pole();
};
#endif
```

---

plik klasa.cpp:

```
#include "klasy.hpp"
float figura::pole()
{
    cout << "Brak metody virtual-->";
    return 0.0;
}
float figura::obwod()
{
    cout << "Brak metody virtual-->";
    return 0.0;
}
void figura::name()
{
    cout << "Brak metody virtual-->";
}
kwadrat::kwadrat(float a)
{
    this->a = a;
    nazwa = "kwadrat";
}
void kwadrat::name()
{
    cout << nazwa << ": " << endl;
}
float kwadrat::pole()
{
    return a*a;
}
float kwadrat::obwod()
{
    return 4*a;
}
kolo::kolo(float r)
{
    this->r = r;
    nazwa = "koło";
}
float kolo::obwod()
{
    return 2*3.1415*r;
}
```

```
float kolo::pole()
{
    return 3.1415*r*r;
}
void kolo::name()
{
    cout << nazwa << ": " << endl;
}
trojkat::trojkat(float a, float h)
{
    this->a = a;
    this->h = h;
    nazwa = "trójkąt";
}
float trojkat::pole()
{
    return 0.5*a*h;
}
void trojkat::name()
{
    cout << nazwa << ": " << endl;
}
```

5 Szablon funkcji

Przeciążanie funkcji związane jest związane z koniecznością zapisania kodu każdej wersji funkcji. Może to prowadzić do znacznego wzrostu długości kodu. W wielu przypadkach rozwiązaniem było by zapisanie adaptującego się do typu danych kodu funkcji.

Takie rozwiązanie może zapewnić **szablon funkcji**. Kod funkcji należny poprzedzić deklaracją szablonu typu lub typów na których oparty będzie kod. Warunkiem koniecznym poprawnego zastosowania szablonu jest aby przynajmniej jeden z atrybutów funkcji był tożsamy z typem zwracanym przez funkcję. Definicję szablonu można zrealizować na dwa sposoby jak pokazano poniżej.

```
template<class typ1, class typ2,...>
template<typename typ1, typename typ2,...>
```

Kod realizowany w funkcji opiera się na typach zdefiniowanych w szablonie. W momencie wywołania funkcji realizuje się przypisanie typów do szablonu podając ich listę lub pozwalając na identyfikację typów na podstawie typów atrybutów funkcji, jak pokazano poniżej.

```
float x = funkcje<float>(3.23);
float x = funkcja(3.23);
```

Poniżej pokazano kod programu opartego na szablonie funkcji.

```
#include <iostream>
using namespace std;
template<typename zm_1, typename zm_2>
zm_1 suma(zm_1 A,zm_2 B)
{
```

```
    return A+B;
}
int main()
{
    float A = 4.56;
    int B = 9;
    cout << "suma(" << A << "," << B << ")=" << suma(A,B) << endl;
    cout << "suma(" << B << "," << A << ")=" << suma(B,A) << endl;
    cout << "suma(" << A << "," << B << ")=" << suma<float,float>(A,B) << endl;
    cout << "suma(" << B << "," << A << ")=" << suma<int,int>(B,A) << endl;
}
```

5.1 Szablon klasy

Podobnie jak w przypadku funkcji można utworzyć szablon klasy. W przypadku klasy w momencie tworzenia na jej podstawie obiektu konieczne jest bezpośrednie określenie typu danych przypisywanych do konstruktora. Przykład implementacji szablonu klasy pokazano poniżej.

```
#include <iostream>
using namespace std;
template<typename zmienna>
class dane{
private:
    zmienna A,B;
public:
    dane(zmienna A, zmienna B)
    {
        this->A=A;
        this->B=B;
    }
    zmienna suma()
    {
        return A+B;
    }
};
int main()
{
    dane<int> X(4,3);
    dane<float> Y(3.21,9.99);
    cout << "Suma(int)=" << X.suma() << endl;
    cout << "Suma(float)=" << Y.suma() << endl;
}
```

Zadania

W oparciu o informacje wykładowe i treść niniejszej instrukcji zrealizować poniższe zadania:

a) zaprojektować klasę umożliwiając zdefiniowanie liczby zespolonej.

- zaprojektować co najmniej trzy konstruktory klasy
- wykorzystując przeciążenie operatora zaprogramować podstawowe działania na liczbach zespolonych
- zmodyfikować kod klasy aby możliwe było operowanie na liczbie zespolonej w której podstawą są wartości typu `<int>`, `<float>` lub `<double>`.

b) wykorzystując polimorfizm dynamiczny napisać program katalogujący zapasy magazynowe.